Plezuro

Release 1.0

Piotr Sroczkowski

2015-12-13

Plezuro is a scripting language.

The main version is compiled to Javascript.

```
$f = {this + first * 2};
$y = f(2, f(5, 9));
[y, y]
```

ROZDZIAŁ 1

Tutorial

This is tutorial of Plezuro programming language.

For whom is it?

If you are a programmer who just knows another programming language (ex. c, Java, Python...), this tutorial will bring you the knowledge necessary to create applications in Plezuro. Nonetheless the experience is what is the most important in programming technologies. So just try to write more and more in Plezuro.

If you are new to programming, this tutorial should be readable for you because it explaines how the stuff works but if you do not understand it, maybe you should learn first the basics of Javascript (it is the most related to Plezuro) or eventually another programming language.

Other useful materials

If you want to learn more, you can read the specification of Plezuro. It is written in a not complicated manner. This is just a view from another site. You can read Plezuro scripts on Github and try to write something similar.

1.1 About Plezuro

1.1.1 Origin

Where does Plezuro come from? Basically its author was not fully satisfied with any existing programming language. So he thought about the inventing of a new one. The name comes from the Esperanto and means 'pleasure'.

Useful materials

On Github:

https://github.com/oprogramador/plezuro_js

Official website:

https://plezuro.herokuapp.com

1.1.2 Basic rules

Plezuro is an imperative, object-oriented, functional, procedural and reflective programming language.

The main ideas are:

1. The code should be possibly short (as long as it is human readable).

2. Very simple syntax.
3. The code should be easy to read for a beginner programmer.
4. The power of the language should be based on the standard library, not its syntax.
5. Explicit is always better than implicit.
6. Everything is a variable.
7. Everything is an object.
8. Multiple inheritance.
9. There is no difference between a module, a class and a namespace.
10. No annotations and other additional syntax - everything is based on the basic syntax of the language.

1.1.3 How to

Currently the main version of the Plezuro is compiled to Javascript. In the future it is planned to compile it to another languages (probably c, c♯, Java, Python, Ruby and PHP, eventually Lisp and Fortran). It exists also an interpreted version implemented in c♯ which is not supported anymore. That version is not compatible in 100% with the mainstream Plezuro. It is something like a prototype of the final product.

Plezuro.js

Using the plezuro.jar (download from https://plezuro.herokuapp.com/downloads/plezuro.jar) executable you can compile a script from Plezuro to Javascript. It works for entire files, not a code embedded in HTML. You can use it within a browser, in the server side (using Node.js), for the mobile development (using Cordova) or any other technology that uses Javascript. Another important thing is to attach the plezuro.js (download from https://plezuro.herokuapp.com/downloads/plezuro.js) library (in HTML for browser and Cordova or using 'require' for Node.js). There is an automated support for all the Javascript libraries because you can use all the global variables from the Javascript. A variable cannot contain the dollar sign in its name in contrary to the Javascript so using jQuery instead of a dollar sign you should use the variable jQuery or eventually `eval('$')`.

Basic usage of the compiler: `plezuro.jar input.plez output.plez.js`

1.2 Basics

1.2.1 Hello world

'Hello world' in computing means the simpler application in a specific programming language. It shows generally how a language looks.

There are several ways in Plezuro to write it. It is a scripting language so there is required no main function nor class. The simpler script just returns a value. You can also use the 'dumpl' method for printing to a buffer (in most cases the console).

```
'Hello world!'
```

```
'Hello world!'.dumpl
```

1.2.2 Variables

Probably the most basic feature of a programing language are the variables. What is it? The variable is a block of a program memory that you can change at the runtime. You can assign it to a symbol and it works like the mathematics. The only difference is a variable can change his value. For example `$x = 2 + 5; $y = x * 2`.

In the Plezuro like in the majority of the programming languages a variable name can begin with a letter or an underscore _ and the next characters of the name can be a letter, an underscore or a digit. The case of the characters matters (like in c, Java, Python and so on, differently from SQL and HTML). You can use only the ASCII letters. It is for the readability (it would be extremely strange if someone used Arabic, Chinese characters or some mix of them). The recommended style of the naming all the variables (including functions and modules) is the camelcase (ex. `` `aVeryInterestingVariable` ``).

Declaration

Before we can use any variable, we have to declare it. It is pretty simple, just write the name of the variable and the dollar sign ('$') immediately before it. In next occurrences of a variable, you should write it without the dollar sign.

Scope

The scope of a variable is limited within the curlies (`{, }`) which are used for the function (even a conditional or a loop it is used an inner function). So if you want to use a variable across multiple functions, you have to declare it in a propriate place.

What can be a variable?

In Plezuro there is dynamic typing like in other dynamic languages such as Python, Ruby or Javascript. Also everything is a variable (including functions and modules). So a variable can change its type in the runtime. For example at the first it is a number, then a list and finally a function.

```
// number
$a = 34;

// number - scientific notation
$a1 = 1e34;
$a2 = 2.34e-89;

// number - binary
$a3 = 0b10111;

// number - octal
$a4 = 052;

// number - hexadecimal
$a5 = 0xae;

// string
$b = 'abc';

// list
$c = [a, b];

// associative array
$d = %('a': a, 'b': b)
```

1.2.3 Comments

The comments are like in c++, Java, and c♯. You can comment one line using `//` or multiple lines using `/*` and `*/`.

```
$x = 2 + // This is a comment
21 * 3;
/* Another
  comment
*/
x++;
x
```

1.2.4 Functions

To write a function, you use just the curlies (`{`, `}`). Everything in a curly bracket is a function. It includes also conditionals and loops. The zero argument is accessible via the keyword 'this' and the next ones 'first', 'second' and 'third'. You have also access to the array of the arguments using the keyword 'args'.

The function returns the value of the last statement (like in Ruby). There is no `return` keyword.

```
$f = {
  this * 2 + first + second / third
};
f(1, 2, 3, 4)
```

To count the execution time of a function, use method 'time'. The arguments passed to this method are next passed to the function (first argument becomes zero argument, second argument becomes first and so on).

It returns an object with fields:

- `result` - value returned by the function
- `time` - execution time in seconds

```
$f = {
  this * 2 + first
};
$res = f.time(9, 3);
[res.time, res.result]
```

1.2.5 Built-in methods

In further reading you will take some knowledge of modules and methods in Plezuro. However at this moment you should know how the method works. Basically it is like a function calling. Zero argument (called this object) is accessible via the keyword 'this'. Like in the function you can also use such keywords 'first', 'second', 'third', 'args'. Calling a method, at the first you write the zero argument (this object), than the dot `.`, function name, bracket opening `(`, arguments separated by the comma `,` and the bracket closing `)`. However with no arguments the bracket is not required.

```
$x = 15;
x = x.sin.cos.tan;
x += [x, x, x].length;
x
```

1.2.6 Collections

One of the indispensable features of a programming language are the collections. Of course, Plezuro provides some sorts of it.

List

The most basic one is the list. To create a list, you use square brackets (`[, ]`), the elements are separated by a comma `,`. It can contain objects of different types (like list in Python, Ruby, Javascript or PHP as well List<Object> in Java or List<object> in c♯). It implies that a list can also contain other lists. Moreover a list can contain a self-reference (it means one of its elements is a pointer to this list) because the lists are passed by reference to functions and to collections.

```
$x = 4;
$a = [x, x];
$b = [x, a, 1, ['abc', 56]];
b[0] = b;
b
```

Set

To create a set, you use dollar sign `$` and brackets. This collections is like the mathematical set. Each value can be contained only once.

```
$a = $(3, 4, [2, 9], $('ab', 90));
a.len
```

Dictionary

Basically it is a set of pairs key-value. It is like dictionary in Python, hash in Ruby, Dictionary<object, object> in c♯ or Map<Object, Object> in Java. However there is a notably difference between the Plezuro dictionary and associative array in PHP or object in Javascript because in the dictionary the order of the items does not matter and generally it is stored using a binary tree. You write it with a hash sign # and brackets.

```
$x = 'abc';
$dict = #(
  x: [4, 5],
  'def': 49,
  'ghj': 'ooo'
);
dict.get(x)
```

Associative array

It is like the associative array in PHP or object in Javascript. In the version of Plezuro compiled to Javascript the main appliance of this collection is to pass Javascript objects to methods from libraries. You write it with a percent sign `%` and brackets.

```
$a = %(
  'a': 12,
  'b': 90
);
a['a']
```

1.2.7 Operators

One of the important features of a programming language are operators. Technically it would be possible to create a language without operators. However it facilities much the syntax. For example in an expression `1+3` we have an operator + which does the addition. In comparison with other languages, Plezuro has some special operators such as the comma `,` or the semicolon `;`.

An important issue in Plezuro is you cannot have any operator immediately (excluding whitespaces and comments) before bracket `)`, square bracket `]` and curly `}` close. So after the last element of the list you cannot put the comma and after the last statement of the function you cannot put the semicolon. In such a case it would be a syntax error.

The action of an operator depends of the type of the zero argument. It behaves in the same way like the method call. It is possible to change the operators action in the runtime (even you can cause that 2+2 produce another result than 4 so it is not recommended to change it in abundance).

Generally we can specify some main actions of the operators (ex. for numbers and strings).

Type of left argument	Operator	Action	Example	Result
Number	`+`	addition	`1+4`	`5`
Number	`-`	subtraction	`4-7`	`-3`
Number	`*`	multiplication	`8.5*2`	`17`
Number	`/`	division	`1/4`	`0.25`
Number	`^`	power	`4^3`	`64`
Number	`%`	modulo	`7%3`	`1`
String	`+`	concatenation	`'a'+'b'`	`'ab'`
String	`*`	multiplication	`'a'\*3`	`'aaa'`
List	`+`	concatenation	`[2,3]+[1]`	`[2,3,1]`
List	`*`	multiplication	`[2,3]*2`	`[2,3,2,3]`

```
$a = 21;
$b = [a, a+90];
$c = a * a + b[0];
[a, b, c]
```

There are also some composite operators which are a syntactic sugar, ex. `a += b` means `a = a + b`.

Complete list of composite operators:

- `+=`
- `-=`
- `*=`
- `/=`
- `^=`
- `&=`
- `|=`
- `%=`
- `.=`

Complete list of the operators:

- single-argument operators (the operator at the left, the argument at the right)
  - `!`
  - `&&`
  - `**`
  - `#`
  - `~`
  - `=>`
- single-argument operators (the operator at the right)
  - `++`
  - `--`
- double-argument operators (the zero argument, the operator, the first argument; from the evaluated at the end to the evaluated at the begin)
  - `;`
  - `,`
  - `:=`
  - `=`
  - `+=`
  - `-=`
  - `*=`
  - `=`
  - `^=`
  - `&=`
  - `|=`
  - `%=`

- .=
- ~~
- <->
- <<
- >>
- ?
- |
- &
- <=>
- >=
- >
- <=
- <
- !=
- ==
- !==
- ===
- =~
- !~
- +
- -
- %
- *
- /
- ^
- ^^
- .
- ..
- :

1.2.8 Conditions

At first, you should write the condition (function that return a boolean value). Then write '.if' and in brackets and curlies write the expression that will be executed when this expression is true. It is just a method of the function type. Then you can use methods 'elif' and 'else' which are not required.

```
$x = 21;
{x > 0}.if({
  x++
}).elif({x < 9}, {
  x--
}).else({
  x *= 2
});
x
```

There is moreover another possibility. You can use two operators: `:` which create a pair and `?` which dependently on a boolean value on the left side returns the key or the value of a pair on the right side. In other languages it is one operator `? :` but Plezuro does not have operators divided into two separate tokens.

```
$x = 9;
x = (x < 3) ? 'a' : 'b';       // 'b'
x + ((x < 'e') ? 'a' : 'b')    // 'a'
                               // finally x = 'ba'
```

1.2.9 Loops

Loops look like conditional expressions. You use a method of the function type.

while

At the beginning you write a function which returned value determines if the function being the first argument of the 'while' method is executed. After the first iteration the conditional function is executed again. Execution of the first argument function depends on the value from conditional function and so on.

```
$n = 1;
{n < 1000}.while({
  n = 2*n+1
});
n
```

do

You write the conditional function. The execution of the loop depends on the value returned by this function. So it iterates as long as the function returns true.

```
$i = 1;
{i *= 2; i < 1000}.do;
i
```

each

It is used to iterate a list. In the inner function the zero argument is the index of the current element and the first one is the element.

```
$a = [
  23,
  200,
  20,
  3
];
$s = 0;
$s2 = [];
a.each({
  s += first;
  s2 << this;
});
[s, s2]
```

foreach

The zero argument is the function and the next ones are lists (number of arguments is unlimited). In each iteration the arguments of the inner function are constructed from the index and the elements of these lists at the position of the index. The iteration lasts until we go to the last element of the longest list. If one from the lists has finished, we obtain the `null` value at the appropriate place.

```
$s = [];
{
  s << first + second
}.forEach([9, 2, 4], [10, 20, 1]);
$s2 = [];
{
  s2 << args
}.forEach([2, 9], [90, 11, 30], [0], [], [4, 2]);
[s, s2]
```

times

Use it when you want to repeat a procedure for specific number of times. It is simpler than other loops. The zero argument is the number (of times) and the first one is the function which obtains the iteration number starting from 0 as its zero argument.

```
$ar = [];
20.times({
  ar << this
});
ar
```

1.2.10 Range

To simple iteration over numbers in an arithmetic sequence, Plezuro provides a useful tool which is the range. To create a range, you use the `..` operator. So you can write the begin number, `..` and the end number or eventually the begin number, `...`, the step number, `..` and the end number.

It exists `each` method which iterates the range. In the inner function the zero argument is the index (natural number starting from 0) and the first argument is a number from the range.

```
$a = [];
(1..7..30).each({
  a << args
});
$a2 = [];
(1..4).each({
  a2 << args
});
[a, a2]
```

1.2.11 Import

It is recommended to divide programming projects to multiple source files. You can do it with the 'import' method of the string type. It is also possible to pass same parameters to the imported script, it works in the same way like the function.

```
$a = './bin/js/doc/loop.plez.js'.import;
a
```

1.2.12 Random

In computing it is useful for many purposes to generate random numbers and random strings. For example, when you write a computer player to a game or in a web application to generate registration links, local or web addresses and so on.

In Plezuro it is quite easy. For a random number you do not have to invoke any function or method (however later it is compiled to a method invocation). Just write the 'rand' keyword and you will obtain a number from the uniform distribution between 0 and 1.

To generate a random string, use `String.rand`. The first parameter specifies the length of the result, its default value is 32. The result is composed from letters, digits and underscores.

```
$a = rand;
$b = rand * a + rand;
$f = {
  rand * this
};
$c = String.rand;
$d = String.rand(200);
[a, b, c, d, f(20), rand]
```

1.2.13 Regex

Beside of imperative programming (as it is in Plezuro), there exist also other programming paradigms. It means you can do the same staff in totally different ways. It can sound a little bit complicated but it often makes hard problems easy.

One of these paradigms is the regex (regular expression). It looks like a normal text but there are included some special characters which can mean one of multiple characters, a repetition of a string and so on. It is really useful for web forms validation (ex. postcode, name, date).

Dependently on the engine, regular expressions behave in different ways. Currently Plezuro uses predefined regex engines so in the version compiled to Javascript the rules are the same as in Javascript. You can learn it from http://www.w3schools.com/jsref/jsref_obj_regexp.asp .

The syntax is the following: `r` and the text in single or double quotes. For a single quote inside the single quotes, write it twice and the same in the case of a double quote inside double quotes (ex. `rąbcd"`, `r'ef''gh'`).

For testing a regex (checking if a string matches to a specific regex), use the `=~` operator. (The order does not matter, either string - operator - regex or regex - operator - string.) There exists also an operator which negates it `!~`.

```
$x = r'[a-z]';
$b = x =~ 'abc';
$b1 = 'dfg' =~ x;
$b2 = 'dfg' !~ x;
$b3 = x !~ 'xxx';
$res = [x, b, b1, b2, b3];
res << 'gh-kk' =~ r'[a-z]'                // true
    << 'gh-kk' =~ r'^[a-z]$'              // false
    << 'gh-kk' =~ r'^[a-z]+-[a-z]+$'      // true
    << 'gh-kk' =~ r'^[a-z]{2}-[a-z]{2}$'  // true
    << 'gh-kko' =~ r'^[a-z]{2}-[a-z]{2}$' // false
    << '\\' =~ r'^\\$'                    // true
    << '\\' =~ r'\\'                      // true
    << '\\' =~ r"\\"                      // true
    << '0+9' =~ r"^[0-9]\+[0-9]$"         // true
    << '0+97' =~ r"^[0-9]\+[0-9]$"        // false
    << 'a"k' =~ r"^a""k"                  // true
    << 'ak' =~ r"^a""k"                   // false
    << 'a"k' =~ r'^a""k'                  // true
    << 'a"k' =~ r'^a"k'                   // true
    << "a'k" =~ r'^a''k'                  // true
    << "ak" =~ r'^a''k'                    // false
    << "a'k" =~ r"^a''k"                  // true
    << "a'k" =~ r"^a'k"                   // true
;
res
```

1.2.14 Magic constants

Similar to PHP and Ruby, Plezuro has the magic constants. What is it and why to use it? It is something a little bit like a variable and a little bit like a constant. Its value depends on the place where it is used.

So we have the following magic constants:

- __pos__ - the horizontal position in a line (counting from 0, it is the position where the keyword begins)
- __line__ - the line number in a source file (counting from 0)
- __file__ - the name of the source file
- __dir__ - the real directory of the source file

```
$x = %(
    'pos': __pos__,
    'line': __line__,
    'file': __file__,
    'dir': __dir__
);
x
```

1.2.15 Exceptions

One of the powerful features of a programming language is the exception handling. It is also possible in Plezuro. However the syntax is a little bit different because in this language almost all is based on the methods (even conditionals and loops).

```
$x;
$y;
{
    x = wfwfwfe
}.try({
    x = this['message']
});
{
    Error.new('an error').throw
}.try({
    y = this['message']
});
[x, y]
```

1.3 Objective programming

Plezuro is an object-oriented language, there is multiple inheritance. It is more dynamic then Java and .NET because you can add, change and remove methods at the runtime.

1.3.1 Modules

Modules are in the same time classes and namespaces. You can use a module to create an object, you can use it statically (like static fields and methods in Java) and you can assign other modules as static fields creating namespaces. Of course you can create multiple classes in the same source file, however it is recommended to write exactly one module in one file.

To create a module, you write 'Module.create' and you pass one argument to this method which is an associative array containing module name (field 'name'), module namespace (field 'namespace', the methods (field 'methods'), the parent modules (field 'parents') - the modules from which our module inherits the methods. There are also some other fields that you can pass.

You can create an object of a module using the 'new' method.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'do' : {
            @a * 2 + first
        }
    )
));
$Student = Module.create(%(
    'name': 'Student',
    'parents': [Person]
));
$s = Student.new(%('a': 14));
s.do(5)
```

1.3.2 Objects and methods

The main purpose of creating a module is to use it as a template to creating objects and to call methods from the objects. When you call a method, the object is accessible via the keyword 'this' and like in the function invocation you can use the keywords 'first', 'second', 'third' and 'args'.

Object fields

To access an object field, you use the at sign `@` and the field name. The fields can be created in the constructor when the object is created or later in any method.

Method from method call

To call a method from another method, you write the 'this' keyword, the dot `.` and the method name. Like in the methods calling from outer objects you can omit the bracket when it is no arguments.

Be aware that when you have an inner function (ex. a loop), the keyword 'this' has a different meaning. It is the zero argument of the inner function. In the same way the object fields work.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'say' : {
            'I am '+@name+', '+@age+' years old'
        },
        'long_say': {
            'Good morning, '+this.say
        },
        'older': {
            @age++
        }
    )
));
$adam = Person.new(%('name': 'Adam', 'age': 20));
adam.older;
[adam.say, adam.long_say]
```

1.3.3 Constructor

Technically it would be possible to create an object without constructor. You could initialize all the fields with one or multiple methods (using the builder pattern). Eventually after that you could freeze the entire object or some of its fields.

However for readability of the source code, it is a good idea to use constructors. What is exactly the constructor? It is a method that is invoked immediately after the memory allocation for the object. It is very explicit in Objective C when you at first allocate the memory and later you initialize it (for most cases it is in the same line).

In Plezuro the constructor is a method with the name 'init'. Such a method is called automatically after the object creation.

Default constructor

However when you do not write any constructor, there exists a constructor that is invoked. It is called the default constructor. What does it exactly do? It invokes the constructors of all the parent modules. When a module does not have any explicit parents it inherits from the BasicModule so in such a situation the constructor of the BasicModule is called - it creates the fields of the object from the associative array which is passed as the first argument.

```
$Person = Module.create(%(
  'name': 'Person',
  'methods': %(
      'init': {
          @age = 0;
          @name = first
      },
      'say': {
          'I am '+@name+', '+@age+' years old'
      }
  )
));
$adam = Person.new('Adam');
adam.say
```

1.3.4 Operators

Similar to defining the method of a module, you can define actions that will be done when an object interacts with a specific operator.

```
$Vector = Module.create(%(
    'name': 'Vector',
    'methods': %(
        'list': {
            @list
        },
        '+': {
            $result = [];
            $i = 0;
            $a = this;
            $b = first;
            {
                result.push(a.list[i] + b.list[i]);
                i++;
                i < a.list.len
            }.do;
            Vector.new(%('list': result))
        }
    )
));
$a = Vector.new(%('list': [3, 4, 9]));
$b = Vector.new(%('list': [6, 0, 9]));
a+b
```

1.3.5 Inheritance

To write an object-oriented code except of classes (in Plezuro modules) it is needed to use inheritance and polymorphism. In Plezuro there is multiple inheritance like in Python, it is also similar to multiple inheritance in c++ with virtual method binding. Why multiple inheritance? It is just more similar to the real life. For example, the dog is in the same time a pet and a carnivore.

How does it work? Is is pretty simple, when you invoke a method on an object, if there exists such a method directly in the module of this object, this method is called. Otherwise it is a method from one of the parents of the module (the order of finding a method is the same as the order of declaring the parents in the module. The algorithm of finding a method is recursive. When no method is found, an exception is thrown. It is as you can see it. However in reality in the moment of the module creation all the methods are binded (direct ones and inherited ones), it is for the performance.

What about the polymorphism? Plezuro like other dynamic languages has a duck typing. So you can invoke a method with a specific name from an object, not knowing the module of the object. On the other hand, you can use the multiple inheritance. You can create a module with a specific method and you can use objects of this module in other methods or functions.

Calling a parent method

One of useful features is the possibility of the calling a parent method. The syntax is similar to the Python because in the case of multiple inheritance, you have to specify from which parent you want to call a method. It is also possible to call a method from a super-parent and even from any other class because of duck typing. You write a parent module name, the double colon `::`, the method name and the 'this' keyword as the zero argument.

```
$Animals = Module.create(%(
    'name': 'Animals'
));
$Animal = Module.create(%(
    'name': 'Animal',
    'namespace': Animals,
    'staticFields': %(
        'nr': 0
    ),
    'methods': %(
        'init': {
            Animal::nr++;
            @age = 0
        },
        'say': {
            'I am '+@age
        },
        'older': {
            @age++
        }
    )
));
$Dog = Module.create(%(
    'name': 'Dog',
    'parents': [Animal],
    'namespace': Animals,
    'methods': %(
        'types': {
            ['a', 'b', 'c']
        },
        'say': {
            Animal::say(this)+', dog'
        }
    )
));
$dog = Dog.new;
[Animals::Dog['className'], Animals::Animal['className'], Animal::nr, dog.types[0], dog.say]
```

1.3.6 Static fields and methods

Except of using a module like a template to creating objects, you can use it in a static way with static fields and methods. You have to remember that the static fields and methods are not inherited. Each static field or method is binded strictly with one module.

```
$Machine = Module.create(%(
    'name': 'Machine',
    'staticFields': %(
        'state': 0
    ),
    'staticMethods': %(
        'change': {
            @state = @state.sin + @state.cos
        },
        'doubleChange': {
            this.change;
            this.change
        }
    )
));
Machine.doubleChange;
Machine.doubleChange;
Machine::state
```

1.3.7 Dynamic module change

Method addition/change

Except of creating a method during a module creation you can add dynamically new methods to a module later. We could say 'at the runtime' but this term is not suitable to the Plezuro because in this language everything is at the runtime. The new method is visible in the direct objects of this module (even created before the addition of a new method) as well in the objects of the child modules. You can achieve that using the method 'addMethod'.

Method removing

It is even possible to remove a method. Nonetheless do not do it without any reason. To do it, use the method 'removeMethod'.

```
$Person = Module.create(%(
    'name': 'Person',
    'methods': %(
        'do' : {
            @a * 2 + first
        }
    )
));
$Student = Module.create(%(
    'name': 'Student',
    'parents': [Person]
));
student = Student.new(%('a': 90));
$x = student.do(1, 3, 4);
Person.addMethod('say', {
    'person a='+this['fields']['a']
});
$y = student.say;
Student.addMethod('say', {
    'student a='+this['fields']['a']
});
$z = student.say;
Person.addMethod('say', {
    'person2 a='+this['fields']['a']
});
$z1 = student.say;
Student.removeMethod('say');
$z2 = student.say;
Person.removeMethod('say');
$msg;
{
    student.say
}.try({
    msg = this['message']
});
[x, y, z, z1, z2, msg]
```

1.4 Web browser programming

As you just know, Plezuro is compiled to Javascript. So we can write applications runned in the web browser (client-side web apps).

What should you know first?

To be able to develop web apps in Plezuro, you should know HTML and CSS (at least the basics). You can read it from the following tutorials:

- http://www.w3schools.com/html/default.asp
- http://www.w3schools.com/css/default.asp

1.4.1 How to

Contrary to Javascript, Plezuro cannot be embedded to HTML. It is not a missing feature, rather a way to maintain a readable code. So you must include a Javascript generated from Plezuro source in an HTML file. Moreover you must include the plezuro.js library.

basic/main.plez

```
window.alert('blabla')
```

1.4.2 Output

You can use the standard Javascript ways to output:

- printing into an alert box, using `window.alert()`
- printing into the HTML output using `document.write()`
- printing into an HTML element, using `element.innerHTML=`
- printing into the browser console, using `console.log()`

However recommended is the Plezuro way using `dumpl` method.

It looks like Javascript but be aware ex. `alert()` without `window` will not work unless you pass the `window` variable as the zero argument. This is because the zero argument in Plezuro is treated like this object.

```
window.alert('aa');
document.write('bb');
alert(window, 'cc');
document['body']['innerHTML'] = 'dd';
console.log('ee');
'ff'.dump1
```

1.4.3 HTML elements

In HTML programming (not each web browsew programming is HTML programming because you can use eventually Java applets, Adobe Flash or Silverlight which are deprecated technologies) almost all you see in the browser are the HTML elements. The exceptions include alerts and different sorts of info from browser.

This is a powerful tool. In the desktop application development in most of frameworks you cannot use such tool and it makes the front-end development really harder.

HTML DOM

It stands for HTML Document Object Model. It represents a tree of objects.

What is an HTML element?

It can be an atomic part of the view you see in the browser (ex. an input, a button, a span), as well something you cannot see (because it is hidden, it has the 0 size or it contains no text) or a tree of other elements.

What is a tree?

We can imagine an object tree like a real tree. Each element (node) is a place where one branch is divided to multiple branches (in a special case still one), a leaf (where the tree ends) or the begin of the tree (where it grows out of the ground).

Relations in a tree

One element for another can be:

- child (in special case the first child)
- parent
- sibling (in special case the next sibling)
- ancestor
- offspring
- none of the above

Let us notice a child is also an offspring, as well a parent is an ancestor.

If you just know same basics of HTML, you should know the outer element is HTML. In reality there is a child of the document which is the most outer element in the whole HTML DOM.

Why is it so important?

Knowing an element, you can operate on his children, siblings, parent. You can replace the children order, copy a subtree and so on.

What Plezuro can do?

- change HTML elements (ex. the text)
- add HTML elements
- remove HTML elements
- clone HTML elements
- change HTML attributes
- add HTML attributes
- remove HTML attributes

Listing 1.1: basic/index.html

```
<html>
<body>
<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>
</body>
</html>
```

- change CSS styles
- create new HTML events

It implies some additional features. For example cloning, removing and adding an element, you can move it.

1.4.4 HTML DOM methods and events

After previous chapters you are able to create any HTML document statically with `doument.write`. But probably you wonder to create an interactive app. For example after button pressing an input will change its value.

How to get an element?

You can use the standard Javascript methods in Plezuro. Eventually you can use some methods from external libraries such like jQuery.

document.getElementById

It returns the element with matching 'id' attribute. Remember if you include your script in the HEAD section, the elements are not loaded so you cannot immediately get any element, in this case you should use the proper event.

document.getElementsByTagName

It returns the collection of all the elements of the tag name specified as the first argument. For example `document.getElementsByTagName('div')` will return the collection of all divs in the document.

document.getElementsByName

The collection of all the elements with the value of 'name' attribute specified as the first argument.

document.getElementsByClassName

The collection of all the elements with the value of 'class' attribute specified as the first argument.

document.getElementsByTagNameNS

Similar to getElementsByTagName but the first argument specifies the name of the namespace and the second one the tag name. It is used rather in XML DOM.

document.querySelector

It returns the first element that matches the CSS selector in the first argument. For example `document.querySelector('table button')` will return the first button in the first table that contain a button.

document.querySelectorAll

It returns the collection of all the elements that match to the given CSS selector.

How to operate on elements?

innerHTML

It is the simplest way to change the content of an element, it just inserts any HTML into the element. For example `element['innerHTML'] = '<button>OK</button>'`.

setAttribute

For example `element.setAttribute('name', 'age')`

style

You can change the CSS style. You can use the CSS name with a hyphen - as well the camelcase. For example `element['style']['backgroundColor'] = 'red'`

document.createElement

You can create an element. The first argument specifies the tag name. For example, to create a div, you write `document.createElement('div')`.

document.appendChild

When you have just created an element you can add it to another element either existing in the HTML DOM or not.

document.removeChild

You can remove an element specified as the first argument.

document.replaceChild

You can replace an element passed as the first argument to an element passed as the second argument.

document.createTextNode

You can create a text node with the value specified as the first argument and then append such a node to an element.

How to listen to en event?

To provide some interaction it is useful to listen to the events. We can detect among others: a mouse click, a key pressing, a document being loaded, a document mutation, an object mutation, browser window resizing.

You can see the full list of events for Firefox on https://developer.mozilla.org/en-US/docs/Web/Events, or for any browser on http://www.w3schools.com/tags/ref_eventattributes.asp.

Let us enumerate the most important ones.

onclick

It fires after the mouse click (pressing and releasing the mouse button). For example `element['onclick'] = { this['parentNode'].removeChild(this); null }`

onkeydown

It fires after a key being pressed.

onkeypress

It fires when a key is being pressed (even multiple times).

onkeyup

It fires after a key being released.

dom_methods/main.plez

```
document.getElementById('mainButton')['onclick'] = {
  $a = Number.parseFloat(document.getElementById('a')['value']);
  $b = Number.parseFloat(document.getElementById('b')['value']);
  $c = a + b;
  document.getElementById('c')['value'] = c;
  this['style']['backgroundColor'] = 'rgb('+(rand*256).floor+', '+(rand*256).floor+', '+(rand*256).floor+')';
  null
}
```

1.4.5 Storage

In programming almost all the time you need to store some data. Part of it is more durable and other part less.

In web browser programming you can use the following types of storage:

- Durable unless the page is reloaded or closed:
  - HTML DOM
  - Javascript memory
- More durable:
  - localStorage
  - sessionStorage
  - indexedDB
  - cookies

Naturally, in Plezuro you have the full access to all of them. Moreover there is easy memory sharing between Plezuro and Javascript with global variables or object fields (public as well private).

storage/main.plez

```
document.getElementById('mainButton')['onclick'] = {
  $counter = localStorage['counter'];
  {Object.isUndefined(counter)}.if({
    counter = '-1'
  });
  counter = Number.parseFloat(counter);
  counter++;
  localStorage['counter'] = counter;
  document.getElementById('counter')['value'] = counter;
  null
}
```

1.4.6 jQuery

jQuery is a powerful Javascript library which gives you all the browser features that are not included in the pure Javascript and which makes the code shorter and more readable.

The main features:

- DOM element selection
- DOM manipulation
- events
- AJAX
- control of the asynchronous processing
- extensibility
- multi-browser support

Theoretically jQuery works in the same way in all the browsers. In reality some differences are possible, for example the selector uses the document.querySelectorAll which behaves a little bit differently depending on the browser.

To use this library in Plezuro, be aware of:

- The most common use in Javascript is based on the `$` variable. In Plezuro it breaks the rules of the variable naming so you can access to the jQuery with the `jQuery` variable or eventually using `eval('$')`.
- You must pass the proper variable as the first argument of `jQuery` function, not as the zero argument.
- In Plezuro everything is an object but jQuery selector must be a true Javascript string, so you must invoke the `toString` method.

jQuery/main.plez

Listing 1.2: dom_methods/index.html

```
<html>
<body>
<label for="a">a=</label><input id="a"/>
<label for="b">b=</label><input id="b"/>
<label for="c">c=</label><input id="c" disabled/>
<button id="mainButton">OK</button>


<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.3: storage/index.html

```
<html>
<body>
<label for="counter">counter=</label><input id="counter" disabled/>
<button id="mainButton">OK</button>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.4: jQuery/index.html

```
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
</head>
<body>
<label for="a">a=</label><input id="a"/>
<label for="b">b=</label><input id="b"/>
<label for="c">c=</label><input id="c" disabled/>
<button id="mainButton">OK</button>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

```
jQuery(null, '#mainButton'.toString).click({
  $a = Number.parseFloat(jQuery(null, '#a'.toString).val);
  $b = Number.parseFloat(jQuery(null, '#b'.toString).val);
  $c = a + b;
  jQuery(null, '#c'.toString).val(c);
  jQuery(null, this).css('backgroundColor', 'rgb('+(rand*256).floor+', '+(rand*256).floor+','+
(rand*256).floor+')');
})
```

Fortunately, there is a simpler way to call jQuery in Plezuro using a special binding. Then you use the `_jQ` function and the code looks in the following way:

jQuery_binding/main.plez

```
_jQ('#mainButton').click({
  $a = Number.parseFloat(_jQ('#a').val);
  $b = Number.parseFloat(_jQ('#b').val);
  $c = a + b;
  _jQ('#c').val(c);
  _jQ(this).css('backgroundColor', 'rgb('+(rand*256).floor+', '+(rand*256).floor+','+(rand*256).floor+')');
})
```

1.4.7 AJAX

AJAX stands for Asynchronous Javascript and XML. It is a set of web development techniques to create asynchronous web applications. Nonetheless XML is not necessary in AJAX, it can be any sort of data (JSON, CSV, HTML, CSS, images, Javascript). AJAX is almost always associated with a serwer-side programming (a server response dependent on the request). Technically it is always true because you send a request and a server in regard to it returns a response. However not always a script in a language such as PHP, Python or Ruby is needed. Server can also return some static data (ex. images) according to their path. The best example is the Apache 2 server.

From a client-side developper point of view it enables the communication with the server without page reloading. For example you would like to store some data on the server for security reasons, check if a data provided by the user matches that on the server or download some content dynamically (HTML, CSS, images, other Javascripts) that would be enormous if downloaded fully.

Probably the best way to make an AJAX request is using jQuery at reasons of browser compatibility and code readability.

ajax/main.plez

```
_jQ('#mainButton').click({
  _jQ('#container').load('part.html'.toString);
  null;
  null
})
```

1.4.8 AngularJS

AngularJS is a framework used mostly in single-page applications. It is a part of MEAN software stack:

- MongoDB - a database
- Express.js - a back-end framework
- AngularJS - a front-end framework
- Node.js - a back-end platform

All of these four technologies are based on Javascript so they are enable to develop in Plezuro.

The examples below have origin from https://angularjs.org/ and later were adapted to Plezuro.

angular/todo.plez

Listing 1.5: ajax/index.html

```
<html>
<head>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.3/jquery.min.js"></script>
</head>
<body>
<button id="mainButton">OK</button>
<div id="container"></div>

<script src="../../../plezuro.js"></script>
<script src="main.plez.js"></script>

</body>
</html>
```

Listing 1.6: angular/index.html

```
<!doctype html>
<html ng-app="todoApp">
  <head>
    <script src="https://ajax.googleapis.com/ajax/libs/angularjs/1.4.7/angular.min.js"></script>
    <script src="../../../plezuro.js"></script>
    <script src="todo.plez.js"></script>
    <link rel="stylesheet" href="todo.css">
  </head>
  <body>
    <h2>Todo</h2>
    <div ng-controller="TodoListController as todoList">
      <span>{{todoList.remaining()}} of {{todoList.todos.length}} remaining</span>
      [ <a href="" ng-click="todoList.archive()">archive</a> ]
      <ul class="unstyled">
        <li ng-repeat="todo in todoList.todos">
          <input type="checkbox" ng-model="todo.done">
          <span class="done-{{todo.done}}">{{todo.text}}</span>
        </li>
      </ul>
      <form ng-submit="todoList.addTodo()">
        <input type="text" ng-model="todoList.todoText"  size="30"
               placeholder="add new todo here">
        <input class="btn-primary" type="submit" value="add">
      </form>
    </div>
  </body>
</html>
```

Listing 1.7: angular/todo.css

```
.done-true {
  text-decoration: line-through;
  color: grey;
}
```

```
angular.module('todoApp', [])
  .controller('TodoListController'.toString, {
    $todoList = this;
    todoList['todos'] = [
      %('text': 'learn angular', 'done': true),
      %('text': 'build an angular app', 'done': false)
    ];

    todoList['addTodo'] = {
      todoList['todos'].push(%('text': todoList['todoText'], 'done':false));
      todoList['todoText'] = ''
    };

    todoList['remaining'] = {
      $count = 0;
      angular.forEach(todoList['todos'], {
        $todo = first;
        {!todo['done']}.if({ count++ })
      });
      count
    };

    todoList['archive'] = {
      $oldTodos = todoList['todos'];
      todoList['todos'] = [];
      angular.forEach(oldTodos, {
        $todo = first;
        {!todo['done']}.if({
          todoList['todos'].push(todo)
        })
      })
    }
  })
```

Specification

This is the formal specification of the Plezuro programming language. The purpose is to define the rules that have to be implemented in any version of the language.

For whom is it?

This is for developpers who would like to implement other version of Plezuro, a library, an extension as well to standard users who want to improve their knowledge. The tutorial is more for learning and simple understanding but the specification is more for knowing exactly how Plezuro applications should behave and therefore how compilers and interpreters should behave.

Nonetheless the code is the best way to define exactly the behavior. So each version of Plezuro must match all the automatic tests included in the mainstream implementation.

2.1 Token types

The first step of the compilation is the tokenization.

Plezuro has the following token types (the order matters, when regex or condition of multiple tokens matches, the first of them wins):

Type	Regex or condition	Example
single line comment	\/\/.*	//a comment
multi line comment	\/*.*	/*a comment*/
regex	r((‘([^’]|(‘’))*’)|(“([^”]|(“”))*”))	r’[a-k]’
bracket	\[|\]|\(|\)|\{|\}|(#\()|(\$\()|(%\(	{this + first}
number	‘^(0x[0-9a-f]+)|(0b[01]+)|(0[0-7]+)|([0-9]+(.[0-9]+)?(e[+-]?[0-9]+)?)$’	23.45e56
declaration	\$[A-Za-z_]+[A-Za-z_0-9]*	$aVariable
class field	[A-Za-z_]+[A-Za-z_0-9]*::[A-Za-z_]+[A-Za-z_0-9]*	Person::totalNr
object field	@[A-Za-z_]+[A-Za-z_0-9]*	@name
whitespace	[\t]+	
operator	one of defined char sequences	+
symbol	[A-Za-z_]+[A-Za-z_0-9]*	aVariable
string	(‘’)|(‘.*?([^\\]|(\\\\))’)|(“”)|(”.*?([^\\]|(\\\\))”)	‘a text’

2.2 Whitespaces and comments

The comments behave like they were replaced by a single space. So the code inside is not executed. Neither are the whitespaces executed. Though they have a little impact on the code: it is impossible to insert a comment (nor a

whitespace) inside the symbol (variable) name; the only possibility is to insert comments between multiple tokens.

2.3 Brackets

There exist following types of the brackets:

Opening	Closing	Usage
(	)	ordering of operators
[	]	list creation, element access
{	}	function
$(	)	set creation
#(	)	dictionary creation
%(	)	associative array creation

The general rules between the brackets:

1. Each bracket has to be closed.
2. Closing a bracket without previous opening of it throws an exception.
3. Stack rule: each bracket opening is pushed to the stack, bracket closing causes popping from the stack, when the popped element does not match to the bracket closing, an exception is thrown.

2.4 Constant tokens

A constant token means a token which has a value and its value is set hardly in the code.

Types of constant tokens:

- Number
- String

2.4.1 Number

Each number is floating point. Generally it reflects the mathematical real number.

There exist the following notations:

Name	Regex	Example
Decimal (including scientific notation)	[0-9]+(\.[0-9]+)?(e[\+\-]?[0-9]+)?	1.2e45
Binary	0b[01]+	0b1101
Octal	0[0-7]+	072
Hexadecimal	0x[0-9a-f]+	0xa4f

2.4.2 String

It is a sequence of unicode characters of any length (the only limit is the memory reserved for the application). The delimiter of the string is either single `'` or double `"` quote. The special characters within the string must be escaped with the backslash `\`.

List of special characters:

- \t - tabulator

- \n - new line
- \\ - backslash

2.5 Symbols

A symbol means a name given by a programmer to anything in the program (dependently on the language it can be variable, function, class, structure, union, trait, module, interface etc.). In Plezuro everything is a variable so each symbol represents a variable.

The rules of the variable naming:

1. The first character must be an ASCII letter or the underscore ('_').
2. The next characters must be an ASCII letter, the underscore or a digit.

2.5.1 Declaration

Eeach symbol must be declared at the first use in the source file. The declaration contains the dollar sign `$` and the name of the symbol.

2.5.2 Class fields

Class field (or module field) contains the name of the module, the double colon `::` and the name of the field (the same rules like for the symbol).

2.5.3 Object fields

Object field (or module field) contains the at character `@` and the name of the field (the same rules like for the symbol).

2.6 Operators

The rule to match an operator:

- A single-argument operator (the operator at the left, the argument at the right) is one of the following:
  - "!"
  - "&&"
  - "**"
  - "#"
  - "~"
  - "=>"
- A single-argument operator (the operator at the right) is one of the following:
  - "++"
  - "–"
- A double-argument operator (the zero argument, the operator, the first argument; from the evaluated at the end to the evaluated at the begin) is one of the following:

- ";"
- ","
- ":="
- "="
- "+="
- "-="
- "*="
- "/="
- "^="
- "&="
- "|="
- "%="
- ".="
- "~~"
- "<->"
- "<<"
- ">>"
- "?"
- "|"
- "&"
- "<=>"
- ">="
- ">"
- "<="
- "<"
- "!="
- "=="
- "!=="
- "==="
- "=~"
- "!~"
- "+"
- "-"
- "%"
- "*"
- "/"

- "^"
- "^^"
- "."
- ".."
- ":"

2.7 Syntax errors

Because of the very simple syntax there are very few syntax errors. Even conditionals and loops are created with the methods invoking so syntax errors are possible to occur where the tokens does not match in a proper way.

2.7.1 How to handle syntax errors?

In Plezuro there are no fatal errors. Each error is an exception. When a script containing a syntax error is imported, it should throw an exception. A compiler should create an output script (or a function in a script where the scripts are joint altogether in the output) which throws an exception.

2.7.2 Total list of syntax errors

Name	Occurrence	Examples
BracketStackException	a bracket is not closed, closing of another type of bracket, abundant bracket closing	(2 + 3 (3 + 1 + [)] 4 + 5)
NonExistentTokenException	a token of not existing type	$\alpha\beta\gamma$ = 21
OperatorAfterBracketCloseException	neither operator nor another bracket closing after bracket closing	$x=[2] "oo"
OperatorAfterBracketOpenException	not proper operator after bracket opening or begin of a script	* 43
OperatorAfterOperatorException	an operator after another one (although there are exceptions)	2 + * 5
OperatorBeforeBracketCloseException	an operator before bracket closing	(2 + 3 -)
ValueAfterValueException	a constant token after another one	3 + *

2.8 Variables

Everything is possible to assign to a variable. Each variable must be declared.

2.8.1 Scope

The declaration determines the scope of a variable. It is possible to hide a variable with another variable with the same name.

```
$x = 21;
$y = 30;
$f = {
  $x = this;
  x + 2
};
$a = f(x); // should return 23
$b = f(y); // should return 32
[a, b]
```

2.9 Expressions

An expression is a single command separated with the `;` operator. Although it can be multiple instructions (ex. `x = y = 2 + 4`). What does it mean exactly? From the Plezuro point of view the expression is an atomic instruction but from the CPU's (even purely theoretical CPU) point of view one expression can contain multiple instructions.

2.10 Functions

A function is a set of operations that can be executed in any moment at the runtime. A function takes any number of arguments which are accessible with the following keywords:

- this - zero argument
- first - first argument
- second - second argument
- third - third argument
- args - the list of the arguments

In case of an inner function the arguments of the outer one are hidden by those of the inner one.

Each function returns a value. The type is not specified. They are two ways to return a value:

- explicitly - with the `return` keyword
- implicitly - the last expression is returned

2.10.1 Methods

A method works in the same way as a function. The only difference is the object (variable before the dot sign) is passed as the zero argument and the next arguments are numbered from 1. In method calling when there is no arguments (except for this object) the bracket after the method name is not required.

2.10.2 Scripts

Even the scripts behave in the same manner as a function. They take arguments (the access in the same way) and they return a value.

2.11 Magic constants

Each compiler (and even interpreter) of Plezuro must provide the following magic constants:

- __pos__ - the horizontal position in a line (counting from 0, it is position where the token begins)
- __line__ - the line number in a source file (counting from 0)
- __file__ - the name of the source file
- __dir__ - the real directory of the source file

It is recommended to replace these keywords with constant strings in the output code. There is no recommendation in the intermediate code. __dir__ does not have to return the canonical form of the path (all the subdirectories from the highest level to the lowest (ex. '/home/user/programming/plezuro/my_plezuro_project')) but must return a valid real Unix path (ex. '/home/user/programming/something/project/../../plezuro/my_plezuro_project').

Even for not Unix systems the path must be in the Unix notation (with slashes).

2.12 Modules

A module is in the same time a class, a namespace and a static module. The module determines the type of a variable. Each variable is associated with a module (built-in or created by a Plezuro developer). Therefore the module determines the action in a method or operator calling.

2.12.1 Constructor

Always after an object creation the constructor is called. Its name is 'init'.

2.12.2 Inheritance

A module can inherit from multiple other modules. The relation of the inheritance is transitive not equal. It means a module cannot inherit from itself and an ancestor of a module cannot be in the same time its offspring. The inheritance includes methods and operators (neither static fields nor static methods). It is possible to override what is inherited. When a module does not inherit explicitly, it inherits from the Module.BasicModule which is ancestor of all the modules.

2.12.3 Static fields and methods

Static fields and methods are associated with exactly one module. They are not inherited.

2.12.4 Duck typing

Everywhere is duck typing. It means it is possible to call a method with given name from an object if such a method exists in its module (directly or inherited). Moreover objects can be passed as the zero argument to methods from totally different modules.

Implementation

This is documentation of the mainstream Plezuro implementation.

For whom is it?

This is for developers who want to write extensions for the language, as well for them who want to write their own implementation (for obtaining the exact knowledge how it works and how to create a compiler in a simple way).

3.1 Tools

The basic tool is the Java 1.8. However in the future it is scheduled a compiler implemented in Plezuro.

To fully utilize the source of Plezuro, you need:

1. JRE 1.8
2. JDK 1.8
3. A Unix-like operating system.
4. Realpath
5. Node.js
6. Nesh

In not Unix-like systems the Plezuro development is still possible, although you have to rewrite the automating scripts.

3.2 Main parts

The source code is divided into the following parts:

- node_modules - external libraries providing functionalities above the standard Javascript
- scripts - automated scripts to compiling, running and testing Plezuro
- src/java - source code of the compiler in Java:
- src/js - Javascript libraries providing Plezuro capabilities for Javascript

3.3 General algorithm

We can specify the main parts of the algorithm:

1. Compiling
   (a) Reading the script.
   (b) Checking if the script is not empty (otherwise it returns null).
   (c) Division of the code into the lines.
   (d) Tokenization:
      - lines are divided into tokens
      - except of multiline tokens which can expand to multiple lines
   (e) Eventual changing of token types (ex. from basic ones to its subtokens).
   (f) Validation - detecting of syntax errors by the tokens order.
   (g) Converting of Plezuro tokens into the output language tokens.
   (h) Writing output to the file.
2. Binding output language (currenty Javascript) libraries.

3.4 Scripts

In the 'scripts' directory there are utilities for compilation, running and testing Plezuro.

Name	Action	Example
build.sh	building Java source code into .class files	scripts/build.sh
clear.sh	deleting all generated output files	scripts/clear.sh
deploy_js.sh	compiling single script (into 'bin' directory) for plezuro files, copying for other files, recursive running the same script for the directories	scripts/deploy_js.sh src/plezuro/tests/1.plez
export_jar.sh	exporting .class files into single .jar	scripts/export_jar.sh
make.sh	build.sh, export_jar.sh, copying .jar file into '/usr/bin/', make_js.sh	scripts/make.sh
make_js.sh	make_js_lib.sh, compiling all Plezuro scripts from 'src/plezuro' to 'bin/js'	scripts/make_js.sh
make_js_lib.sh	joining all Javascript libraries into bin/js/plezuro.js	scripts/make_js_lib.sh
test.sh	testing Plezuro scripts	scripts/test.sh

As you can see, some of these scripts invoke others. So to do all you need, just run three of them:

scripts/clear.sh

scripts/make.sh

scripts/test.sh

These three are not combined into one at reason of not mixing their outputs.

3.5 Compiler

The whole compiler is currenty implemented in Java.

It is divided into the following packages:

- mondo.engine - the kernel of the compiler
- mondo.invalidToken - the exception classes combined with syntax errors and their binding to the output language
- mondo.main - the main class and function
- mondo.token - translation of Plezuro specific tokens to the output language

mondo.main and mondo.engine are not dependent on the output language, on the other hand mondo.token and mondo.invalidToken are dependent. So if you want to write a compiler from Plezuro to other language than Javascript, you should change only the packages mondo.token and mondo.invalidToken.

3.5.1 Package engine

This is the kernel of the compiler. It is not dependent on the output language.

Classes:

- Engine - It handles the arguments of the compiler. For each pair input file - output files it invokes the Parser.
- Parser - It reads the input, handles the situation of an empty file, invokes the Tokenizer, it invokes the basic operations for each token and finally it writes to the output.
- Tokenizer - It divides the code into tokens by conditions given in token classes.
- Validator - It detects all the possible syntax errors.

3.5.2 Package invalidToken

This is the set of all exception classes. It includes their mapping to the Javascript. All the other classes inherit from the InvalidTokenException.

3.5.3 Package token

This is the set of all the tokens and subtokens of Plezuro with their translation to Javascript.